

# Arquitectura de Computadores II

LESI - 3º Ano

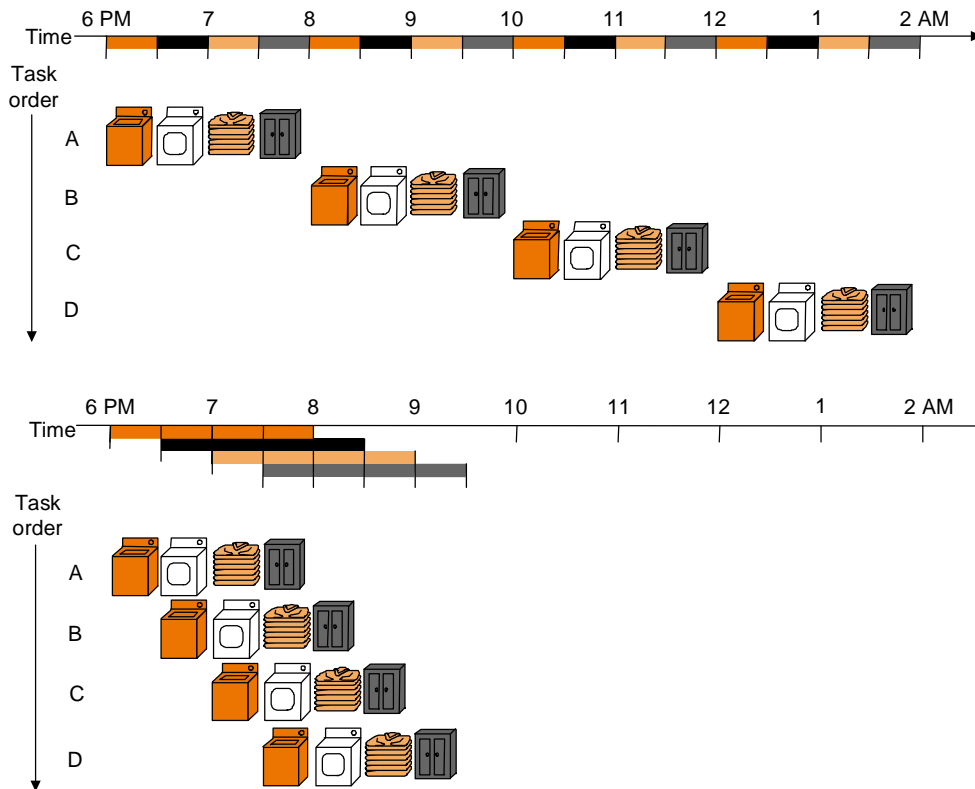
## *Pipelining*

João Luís Ferreira Sobral  
Departamento de Informática  
Universidade do Minho

Janeiro 2002

# Visão global de *Pipelining*

- Técnica que permite sobrepor a execução de várias instruções

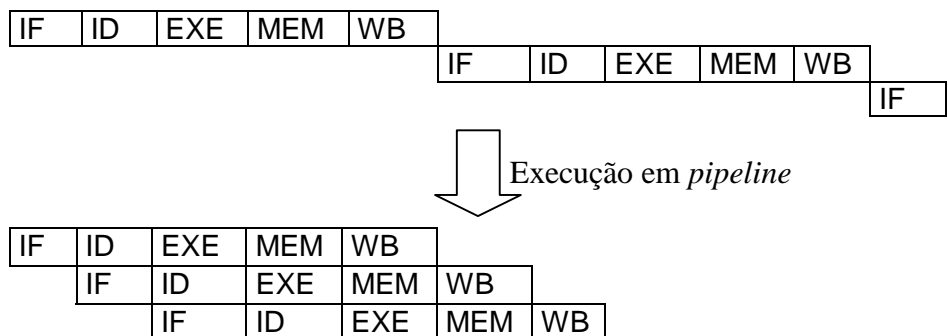


- Cada fase (estágio) da *pipeline* executa de forma concorrente, como numa linha de montagem

- *Pipelining* em MIPS

- Fases de execução das instruções no MIPS:

1. Busca da instrução (IF)
2. Leitura dos registos e decodificação das instruções (ID)
3. Execução da operação ou cálculo de endereço (EXE)
4. Acesso ao operando em memória (MEM)
5. Escrita do resultado em registo (WB)

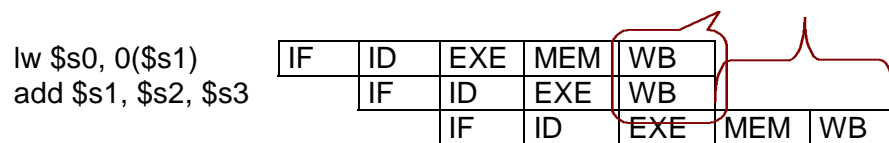


## Visão global de *Pipelining*

- Desempenho do processador com *pipelining*
  - Duração de cada ciclo de execução em *pipeline* é igual à duração do estágio mais longo, adicionando a sobrecarga da *pipeline* (passagem de informação entre estágios)

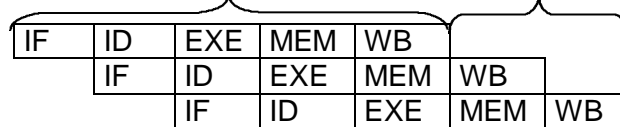
$$T_{cc\ pipeline} = \max [ \text{estágio}_i ] + \text{sobrecarga de } pipeline$$

- A execução em *pipeline* requer que as várias fases estejam balanceadas (i.e., demorem o mesmo tempo a executar):**



- O tempo (ideal) de execução com *pipelining*:

$$T_{execpl} = [ \#estágios + (\#instruções - 1) ] \times T_{cc}$$



- Pipelining* aumenta o débito (instruções realizadas por unidade de tempo) mas não diminui o tempo necessário para cada instrução!**

- O ganho é **potencialmente** igual ao número de estágios da *pipeline*

$$\text{ganho} = \frac{T_{execSc}}{T_{execPl}} = \frac{\#instruções \times \#estágios \times T_{cc}}{[\#estágios + (\#instruções - 1)] \times T_{cc}} \underset{\#instruções \rightarrow \infty}{=} \#estágios$$

- O ganho pode ser interpretado de forma diferente, consoante a base de comparação seja uma implementação *single-cycle* ou multi-ciclo

### base multi-ciclo

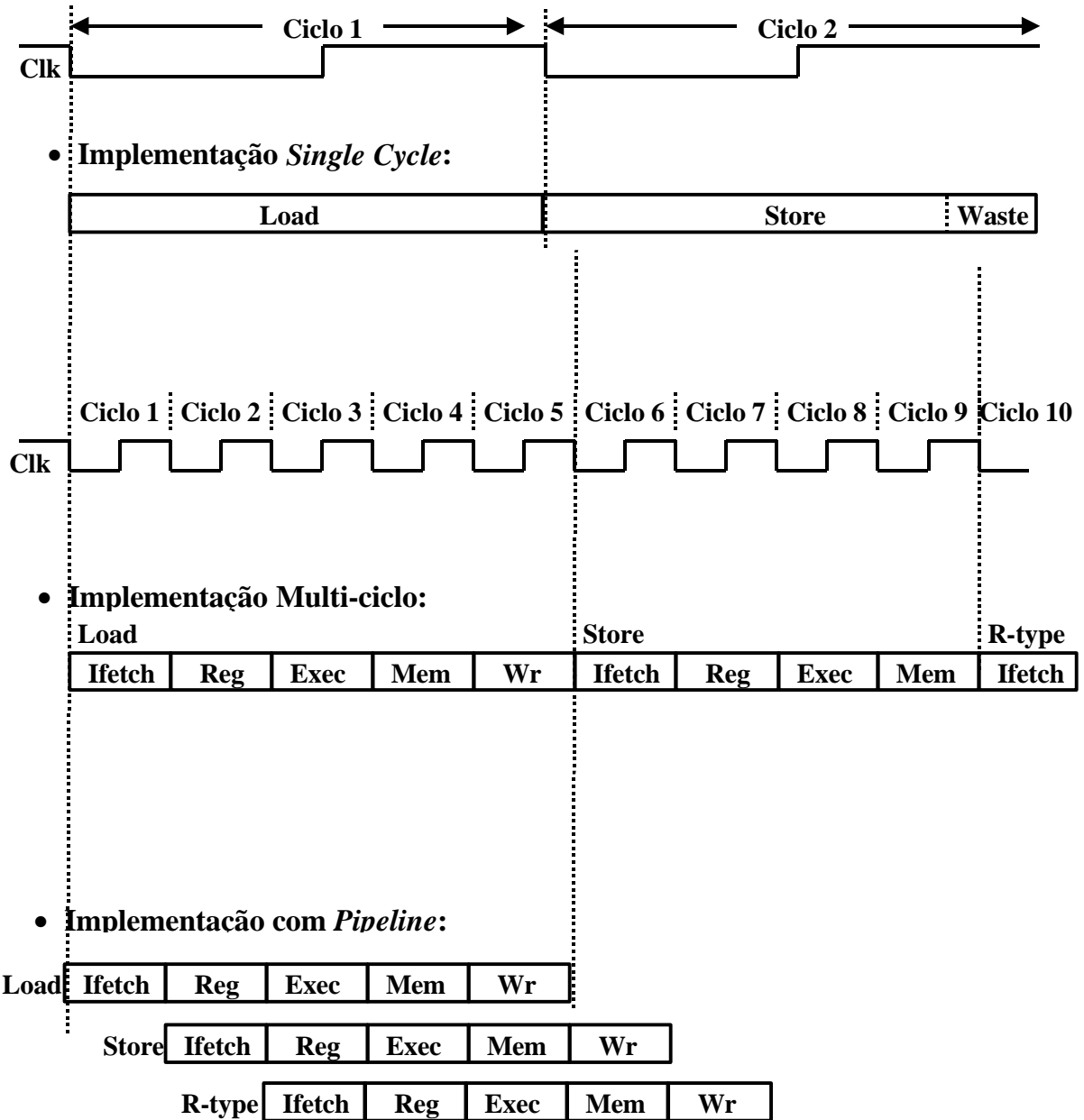
$$CPI_{pipeline} = CPI_{multi-ciclo} / \# \text{estágios da } pipeline$$

### base *single-cycle*

$$T_{ccpipeline} = T_{ccsingle-cycle} / \# \text{estágios}$$

# Visão global de *Pipelining*

- *Single-cycle versus Multi-ciclo versus Pipelining*



## Visão global de *Pipelining*

- Exemplo de desempenho do processador MIPS com *pipelining*

- Tempos acesso às unidades funcionais:

Memória (Leitura ou escrita)	ALU	Banco de registos
2 nanosegundos	2 nanosegundos	1 nanosegundo

- Qual a diferença de desempenho entre uma implementação com *pipelining* e uma implementação *single-cycle* com ciclo de duração fixa?

- Duração de cada instrução:

Tipo de Instrução	Busca da instrução	Leitura de registos	Operação na ALU	Acesso à Memória	Escrita em registo	Total
Tipo R	2	1	2		1	6 ns
<i>Load</i>	2	1	2	2	1	8 ns
<i>Store</i>	2	1	2	2		7 ns
<i>Branch</i>	2	1	2			5 ns

- $CPI_{single-cycle} = CPI_{pipeline} = 1.$

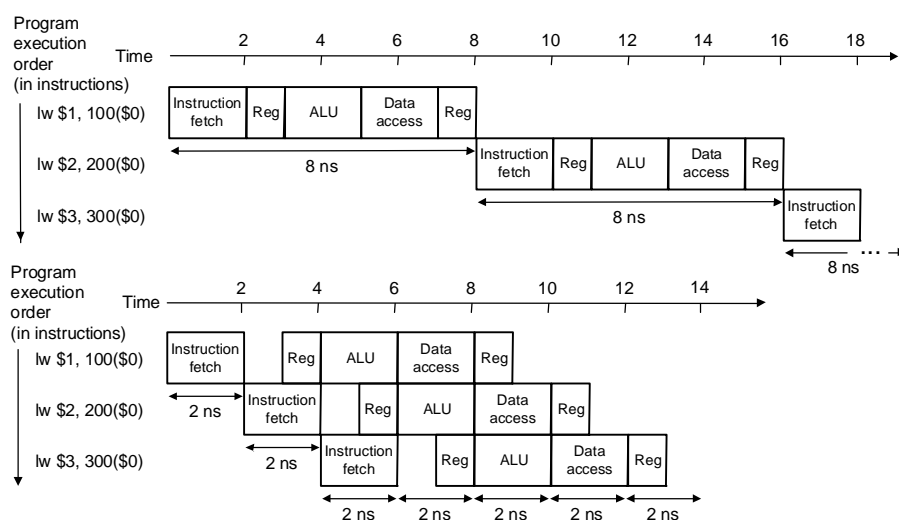
- Duração do ciclo na versão *single-cycle* com o ciclo de relógio fixo:

$T_{csc} = \text{duração da instrução mais longa} = \textit{load} = 8 \text{ ns}$

- Duração do período de relógio com *pipeline*, assumindo 5 estágios:

$T_{cppe} = \text{duração do estágio mais longo} = 2 \text{ ns}$

- Ganho (máx.) =  $T_{csc} / T_{cppe} = 8 \text{ ns} / 2 \text{ ns} = 4,0x$



- Ganho na execução de 3 instruções =  $24 / 14 = 1,74x$

- Ganho na execução de 1000 instruções =  $8000 \text{ ns} / 2008 \text{ ns} = 3,98x$

## Visão global de *Pipelining*

- Projectar o conjunto de instruções para execução em *pipelining*
  - Instruções de dimensão fixa facilitam a busca e a decodificação de instruções
  - Poucos formatos de instruções com os registos fonte especificados sempre nos mesmos campos permitem o paralelismo entre a decodificação e a leitura do valor dos registos
  - Uma arquitectura do tipo *load-store*, onde os operandos em memória só aparecem em *load* e *store* facilitam o balanceamento dos estágios da *pipeline*
  - Operandos alinhados em memória tornam os acessos à memória mais rápidos
- Anomalias na execução em *pipelining* (*hazards*)

- **Impedem que a *pipeline* prossiga a execução normal provocando o aumento do CPI médio para valores superiores a 1.**
- **Estruturais** – o hardware não suporta a combinação de instruções em execução (ex. são necessárias duas unidades para acesso à memória, uma para a busca das instruções outra para os acessos a dados em memória)

Solução =>

    duplicar unidades funcionais ou portas de acesso

- **Controlo** – a execução da instrução seguinte depende de uma decisão baseada num resultado anterior (ex. saltos condicionais)

Soluções =>

    1. empatar a *pipeline*

    2. prever o salto

    3. retardar o salto

- **Dados** – a execução da instrução seguinte depende de dados produzidos por instruções anteriores.

Solução =>

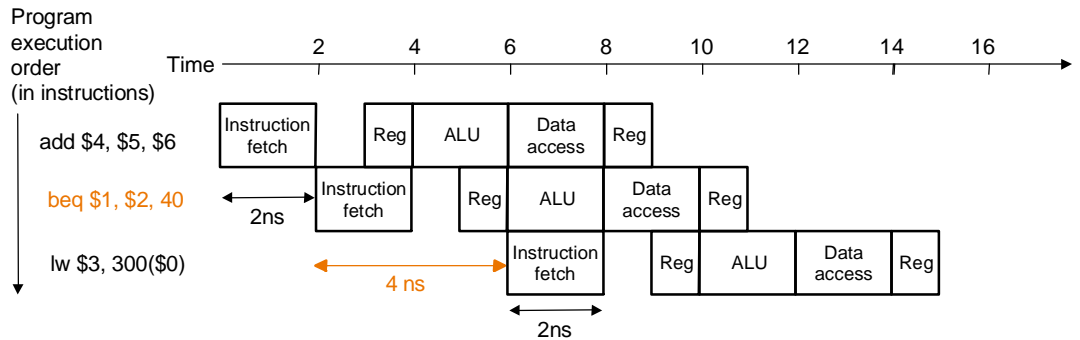
    1. delegar no compilador

    2. encaminhar os dados entre unidades

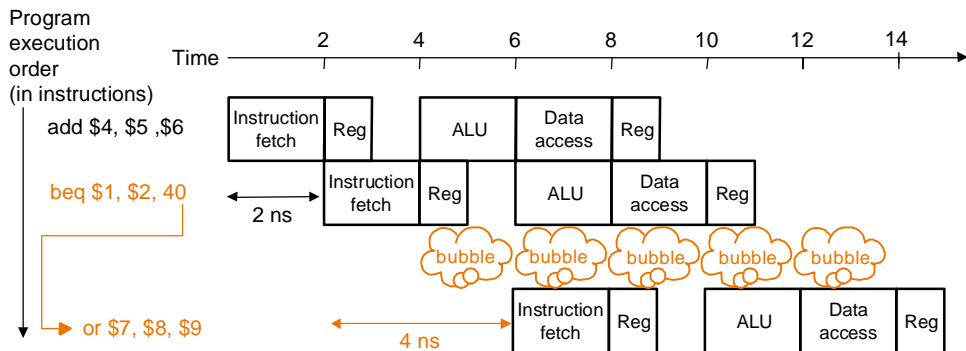
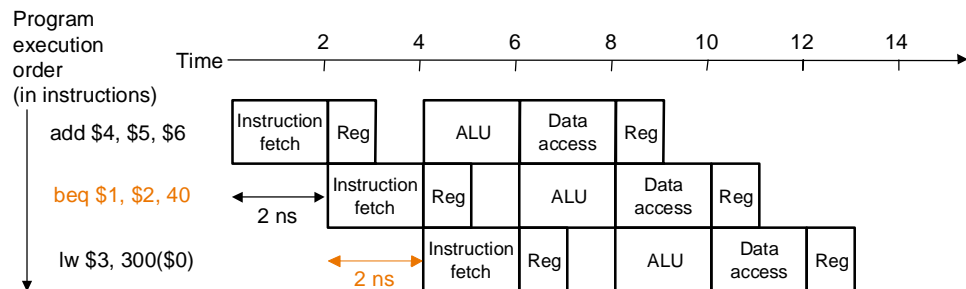
# Visão global de *Pipelining*

- Exemplos de anomalias de controlo na execução em *pipeline*

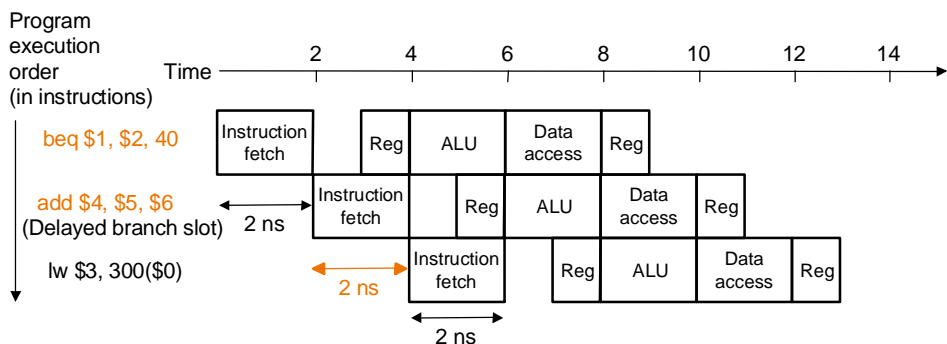
- Empatar a execução de um salto



- Prever o salto como não tomado



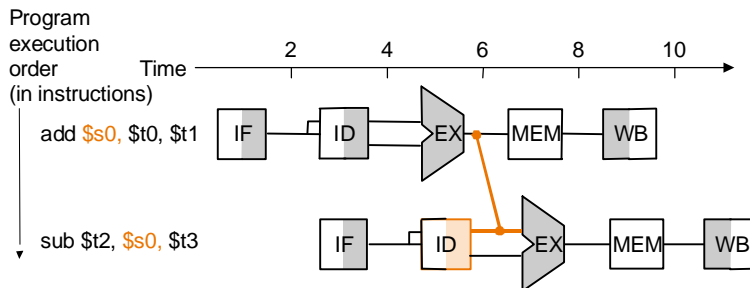
- Execução retardada do salto



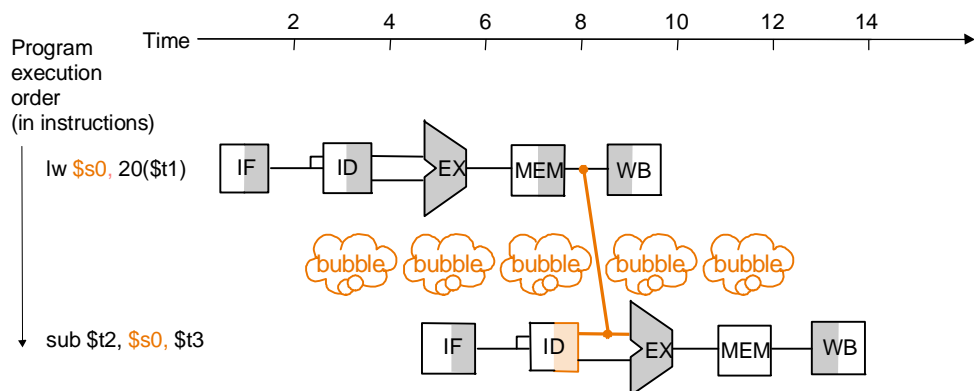
# Visão global de *Pipelining*

- Exemplos de anomalias de dados na execução em *pipeline*

- Em caminhar os dados para resolver a dependência



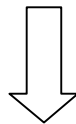
- Limitação no encaminhamento dos dados



- Reordenação das instruções para resolver a dependência de dados

```

# reg $t1 contém &v[k]
lw $t0, 0($t1)      # $t0 (temp) = v[k]
lw $t2, 4($t1)     # $t2 = v[k+1]
sw $t2, 0($t1)     # v[k] = $t2
sw $t0, 4($t1)     # v[k+1] = $t0
    
```



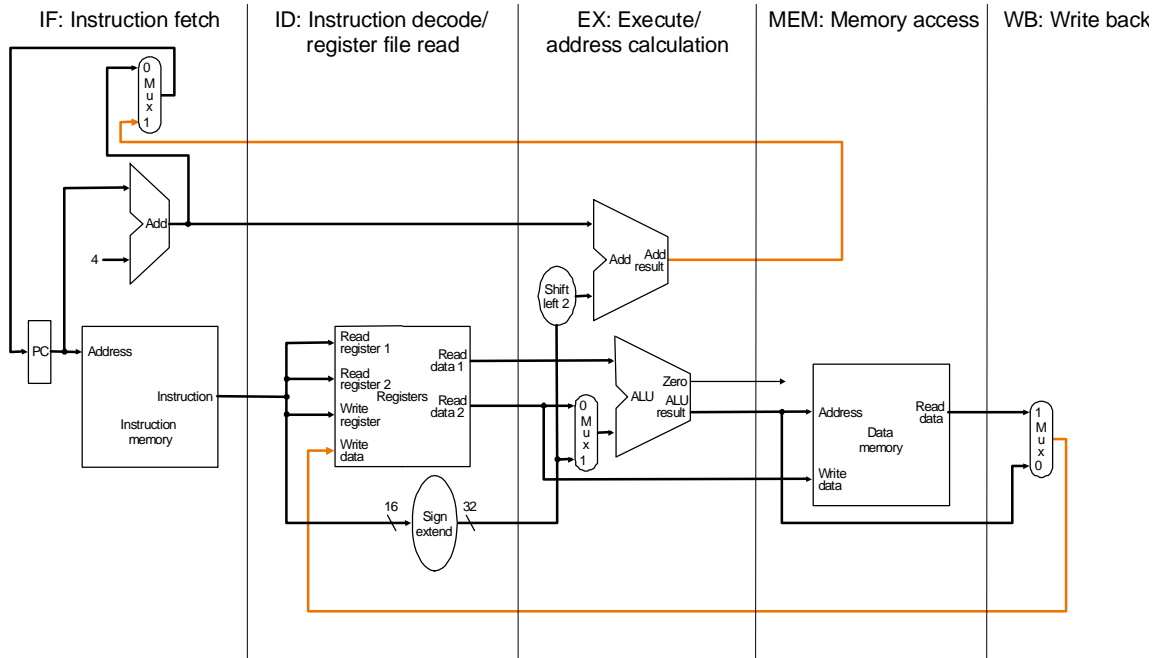
```

lw $t0, 0($t1)     # $t0 (temp) = v[k]
lw $t2, 4($t1)     # $t2 = v[k+1]
sw $t0, 4($t1)     # v[k+1] = $t0
sw $t2, 0($t1)     # v[k] = $t2
    
```

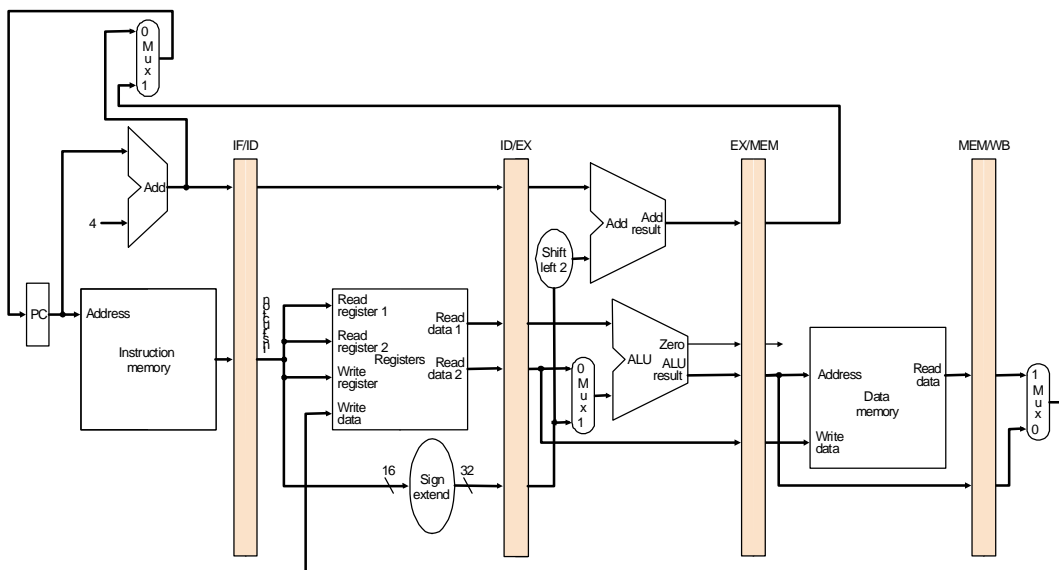


# Pipelining

- DP para suporte à execução em *pipeline*
  - 5 fases de execução => 5 estágios da *pipeline*



- Entre cada fase da *pipeline* é necessário introduzir um registo para conter **toda** a informação necessária à execução dessa instrução
- O nome de cada registo corresponde às fases que separa (Ex. IF/ID)
- O valor dos registos avança simultaneamente com a execução das instruções

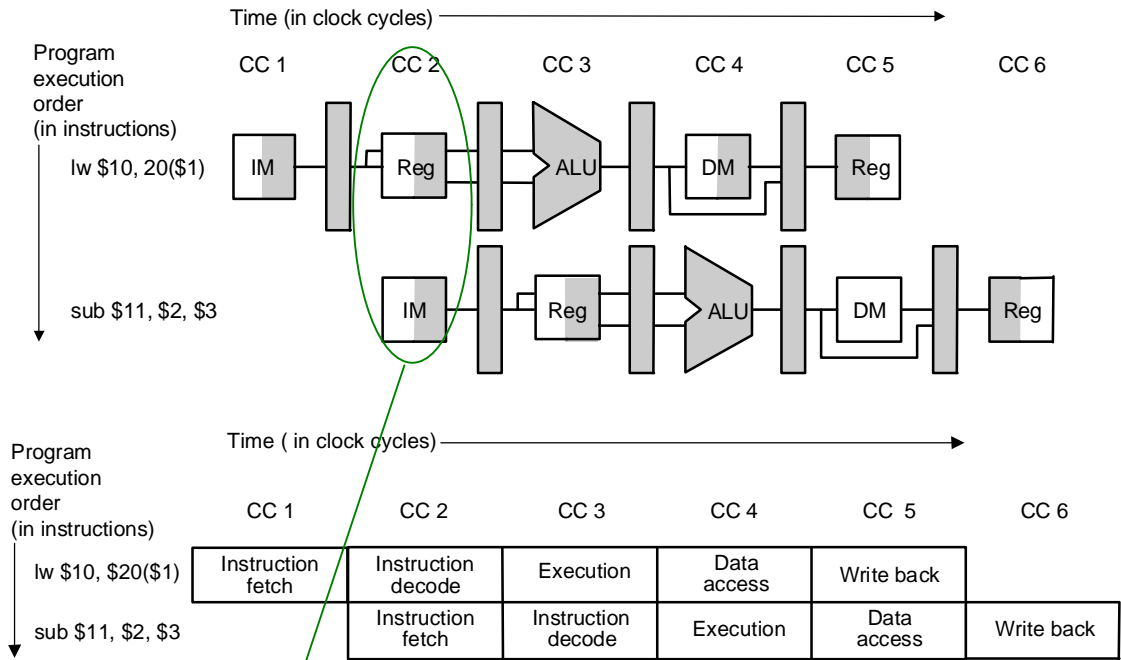


# Pipelining

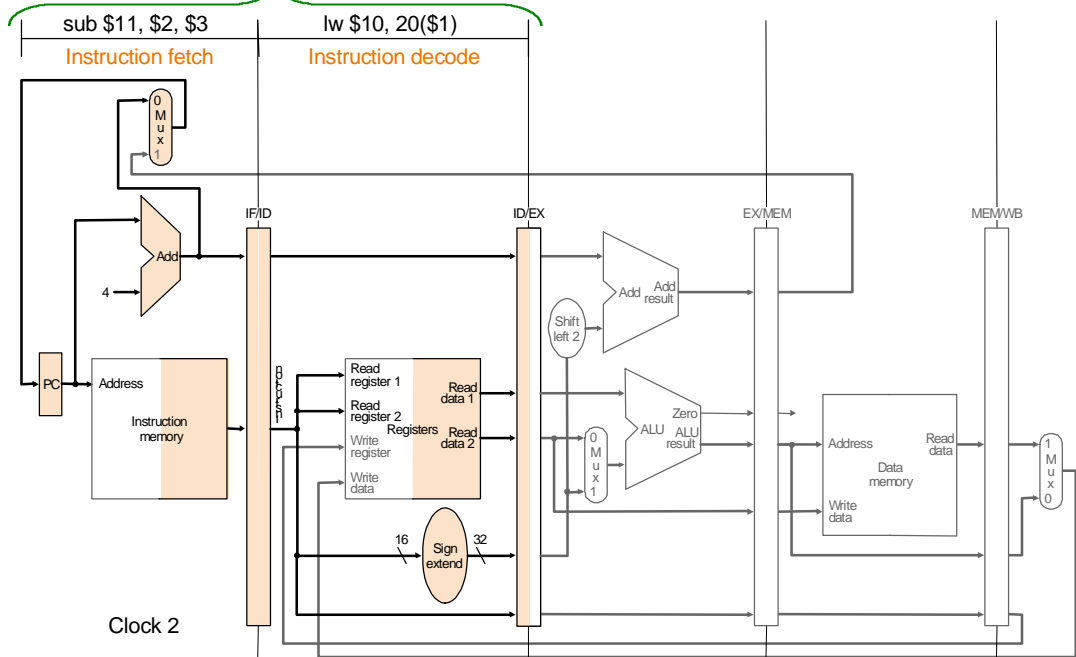
- Representação gráfica da execução em *pipeline*

```
lw $10, 20($1)
sub $11, $2, $3
```

- Representação com um diagrama multi-ciclo



- Representação de um ciclo com um diagrama *single-cycle* (ciclo 2)

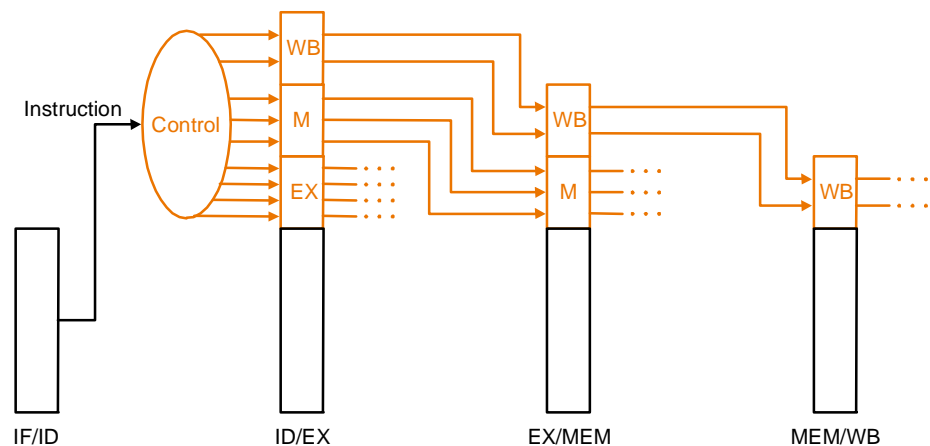


# Pipelining

- Unidade de controlo para suporte à execução em *pipeline*
  - Os sinais de controlo utilizados na versão do MIPS com *pipeline* são idênticos aos da versão *single-cycle*
  - O registo PC e os registos da *pipeline* são escritos com novos valores a cada ciclo.
  - A principal diferença consiste no facto de agora os sinais estarem associados à fase em que são necessários. As primeiras duas fases (IF e ID) são iguais para todas as instruções.

Instrução	EXE			MEM			WB	
	RegDst	ALU op	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
Tipo R	1	10	0	0	0	0	1	0
lw	0	00	1	0	1	0	1	1
sw	X	00	1	0	0	1	0	X
beq	X	01	0	1	0	0	0	X

- A informação de controlo é adicionada aos registos de *pipeline*, sendo gerada na fase de descodificação e acompanha a execução da instrução ao longo da *pipeline*.



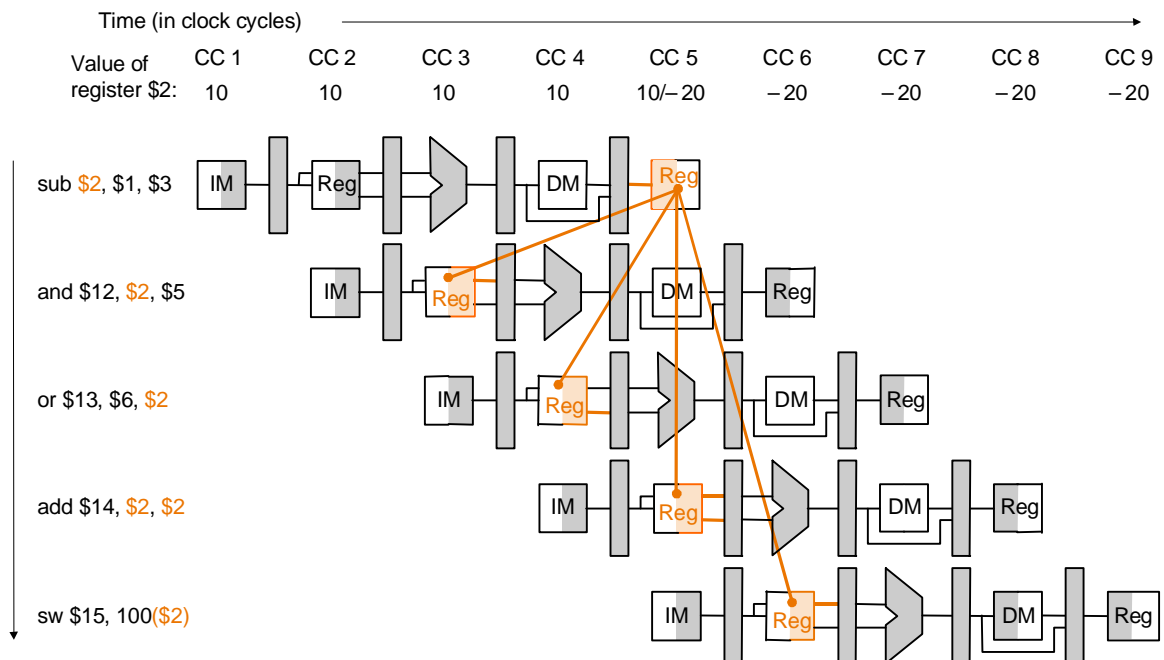
# Pipelining

- Dependências de dados (anomalias)
  - A execução em *pipeline* gera problemas adicionais quando as instruções dependem de resultados produzidos por instruções anteriores:

```

sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
    
```

- As anomalias resultam facto das instruções seguintes efectuarem a leitura do banco de registos antes da instrução escrever no registo:



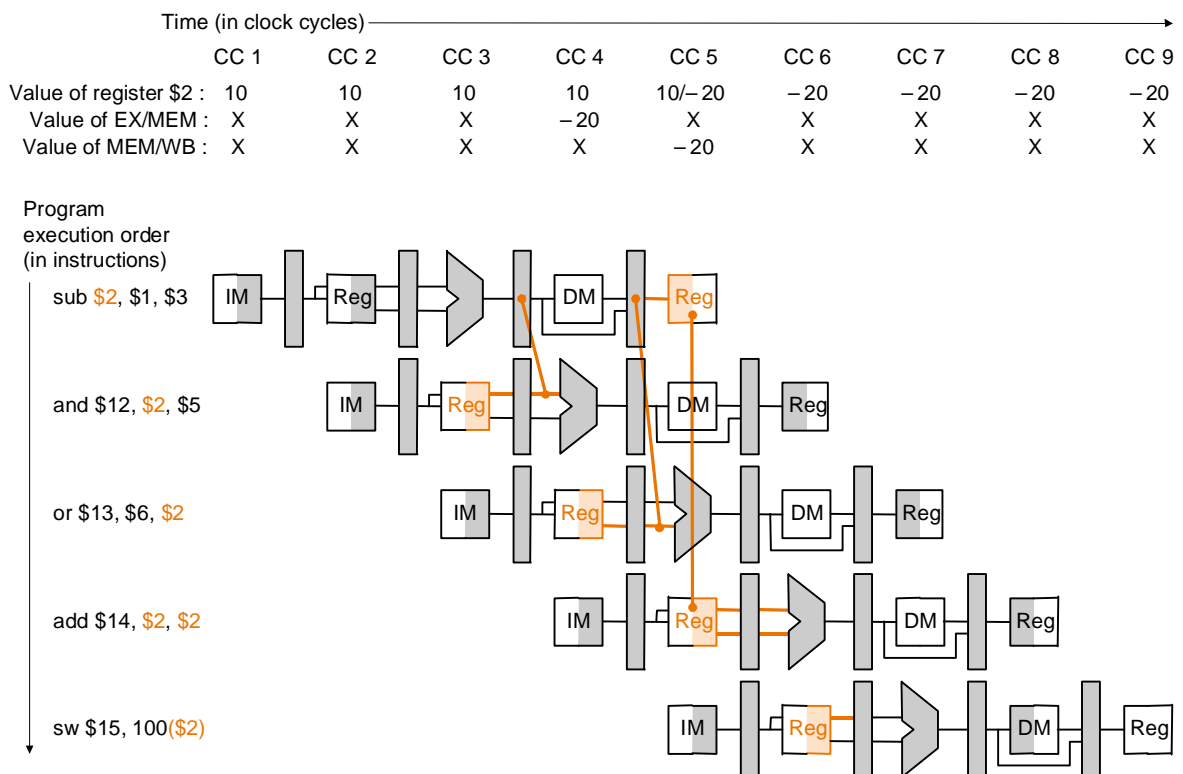
- As anomalias geradas pela leitura e escrita de um registo no mesmo ciclo (p.ex. no ciclo 5) pode ser resolvidas ao nível do hardware se as escritas em registos forem efectuadas na primeira metade do ciclo e as leituras na segunda metade do ciclo.
- Uma solução (pouco eficiente) para os restantes casos seria obrigar o compilador a introduzir instruções de *nop* para eliminar as dependências.

```

sub  $2, $1, $3
nop
nop
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
    
```

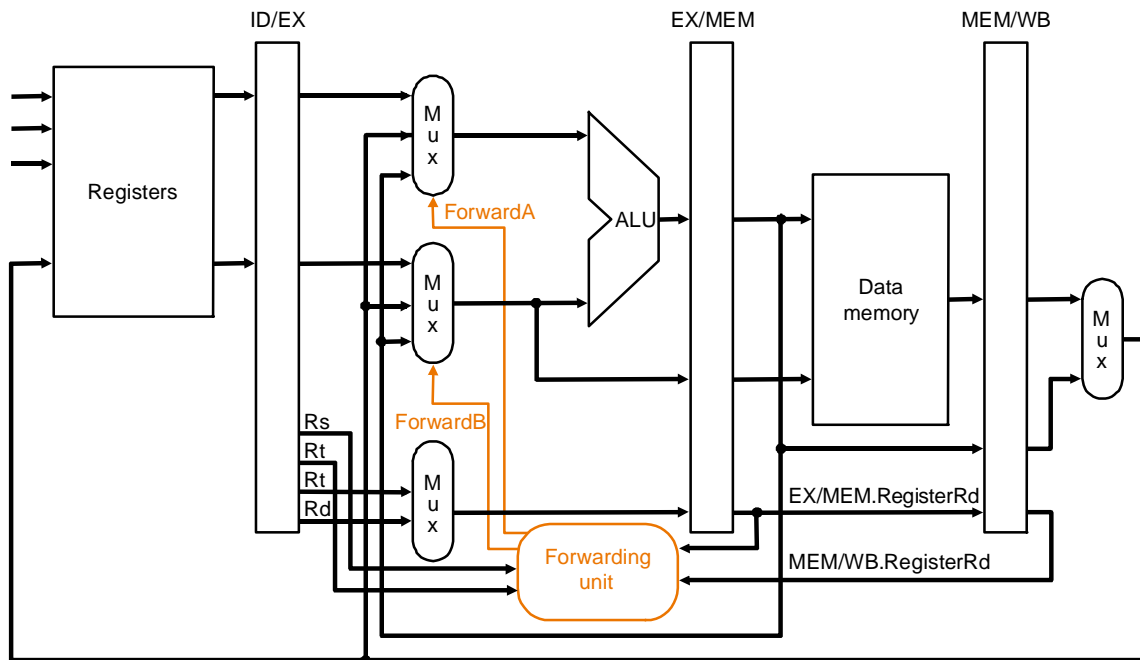
# Pipelining

- Resolução de dependências de dados (anomalias)
  - Deteção das dependências de dados** – detecção dos casos em que o registo destino numa instrução na fase de EX ou WB é utilizado como uma das entradas da ALU (fase EX):
    - EX/MEM.RegisterRd = ID/EX.RegisterRs
    - EX/MEM.RegisterRd = ID/EX.RegisterRt
    - MEM/WB.RegisterRd = ID/EX.RegisterRs
    - MEM/WB.RegisterRd = ID/EX.RegisterRt
  - Encaminhamento de dados** – Na maior parte das dependências de dados o resultado já se encontra calculado (em EX/MEM ou MEM/WB) só não foi ainda escrito no banco de registos. Assim é possível criar atalhos no DP para encaminhar os dados e resolver as dependências.



# Pipelining

- Resolução de dependências de dados com encaminhamento de dados
  - O DP para resolução de dependências deve ter a capacidade de encaminhar para as entradas da ALU os valores em EX/MEM e MEM/WB. Tal é conseguido através da introdução de multiplexers na em cada entrada da ALU



- As condições anteriormente indicadas não são suficientes porque algumas instruções não escrevem nos registos ou escrevem em \$0:

Só efectua o encaminhamento quando a instrução escreve num registo e esse registo não é \$0

```

if (EX/MEM.RegWrite)
  and (EX/MEM.RegisterRd≠0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs) ForwardA = 10
    
```

```

if (EX/MEM.RegWrite)
  and (EX/MEM.RegisterRd≠0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRt) ForwardB = 10
    
```

```

if (MEM/WB.RegWrite)
  and (MEM/WB.RegisterRd≠0)
  and (EX/MEM.registerRd≠ID/EX.RegisterRs)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs) ForwardA = 01
    
```

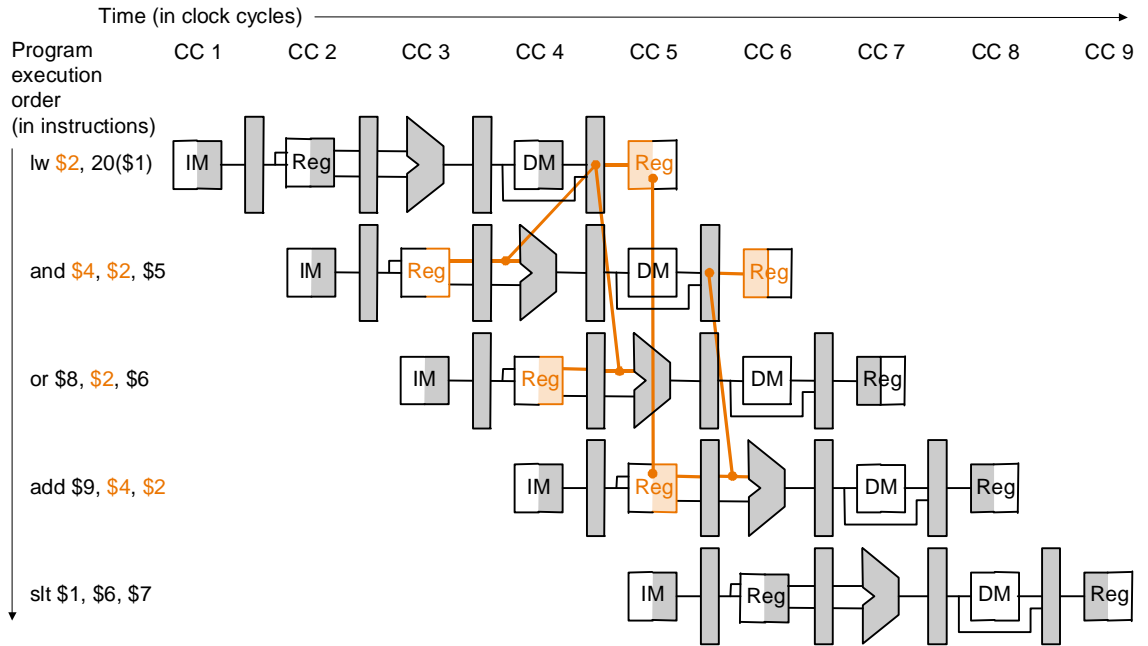
```

if (MEM/WB.RegWrite)
  and (MEM/WB.RegisterRd≠0)
  and (EX/MEM.registerRd≠ID/EX.RegisterRt)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt) ForwardB = 01
    
```

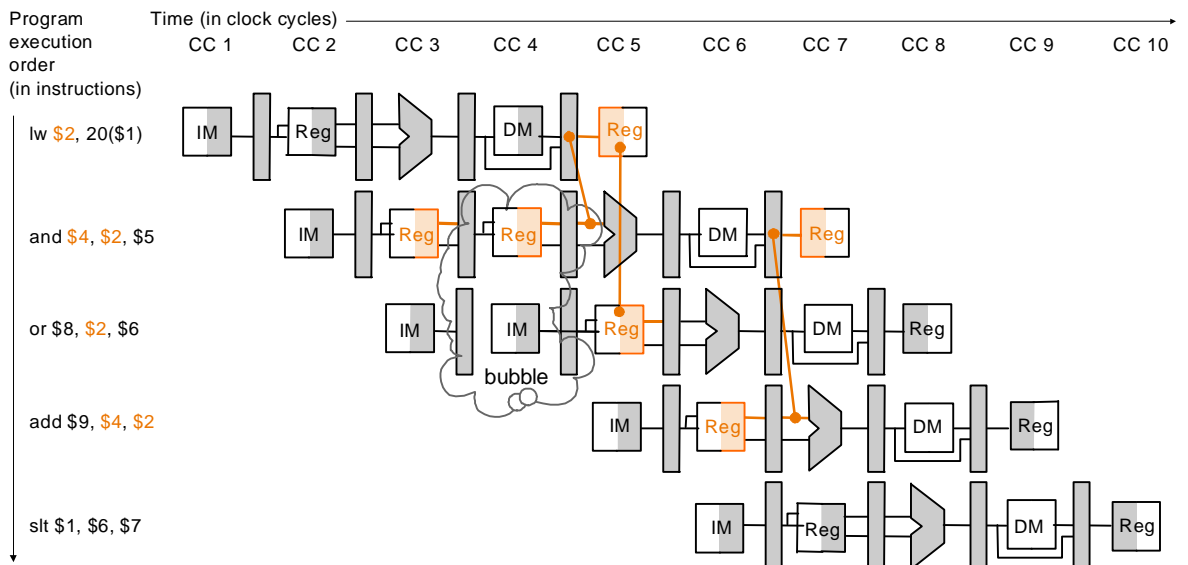
Quando EX/MEM.registerRd = EM/WB.registerRd prefere o valor mais recente (em EX/MEM). exemplo:  
 add \$1, \$1, \$2  
 add \$1, \$1, \$3  
 add \$1, \$1, \$4

# Pipelining

- Resolução de dependências de dados: “empatar” a *pipeline*
  - A utilização do valor de um *lw* na instrução seguinte introduz uma anomalia que não pode ser resolvida com atalhos.



- Nos caso em que as anomalias não são resolvidas é necessário empatar a *pipeline* até que o valor esteja disponível:

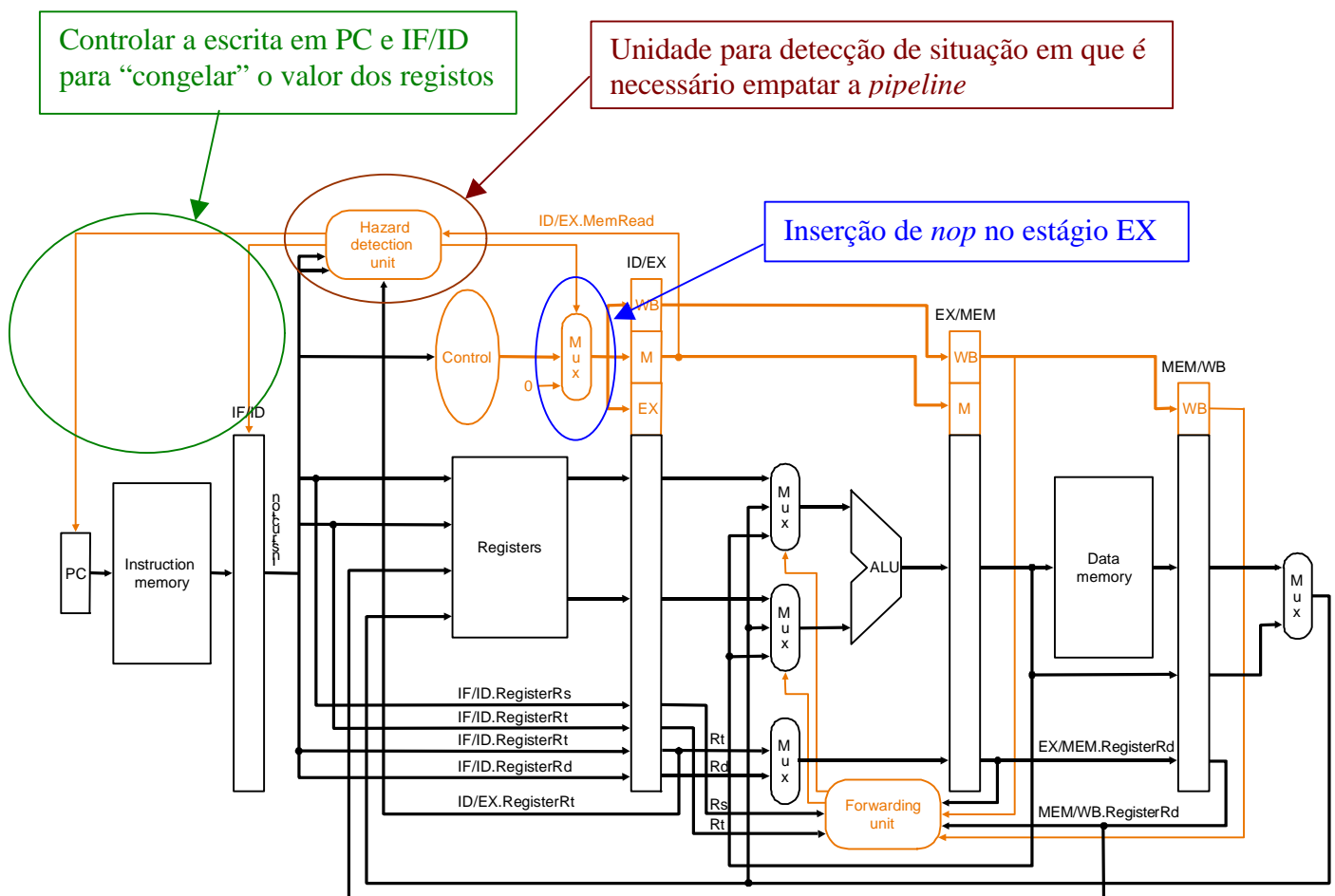


# Pipelining

- Resolução de dependências de dados: “empatar” a *pipeline* (continuação)
  - A condição em que é necessário empatar a *pipeline* é dada por:
 
$$\text{if (ID/EX.MemRead) and}$$

$$\text{( (ID/EX.RegisterRt=IF/ID.RegisterRs)}$$

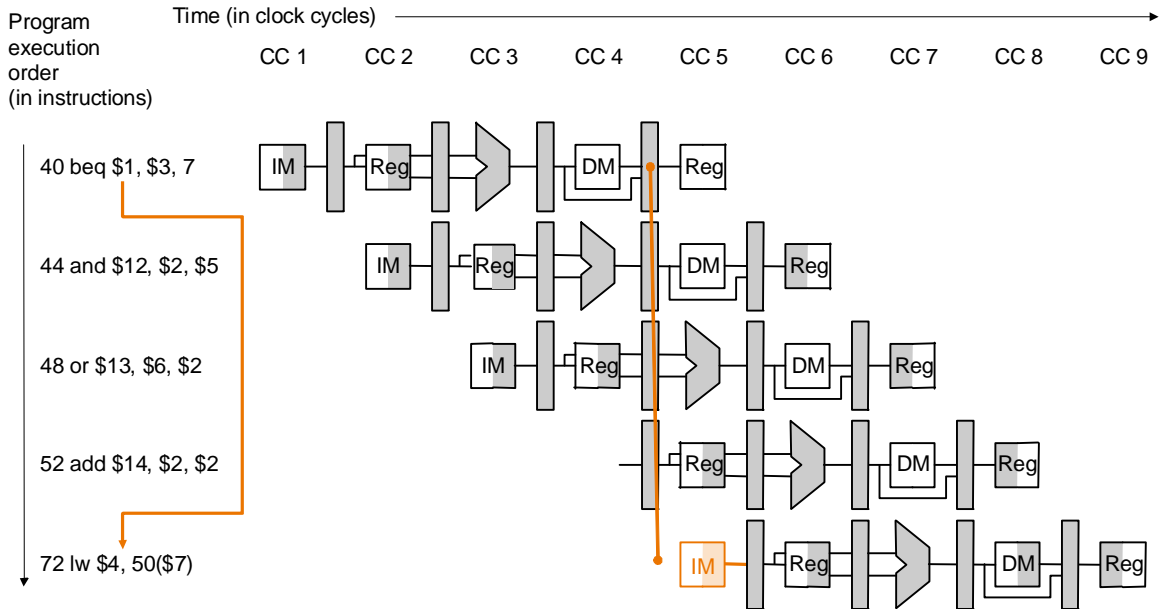
$$\text{or (ID/EX.RegisterRt = IF/ID.RegisterRt))}$$
  - Esta condição pode ser implementada por uma nova unidade. Empatar a *pipeline* é implementado através de uma sinal de não permite a escrita nos registos PC, IF/ID e que insira um *nop* no estágio EX.





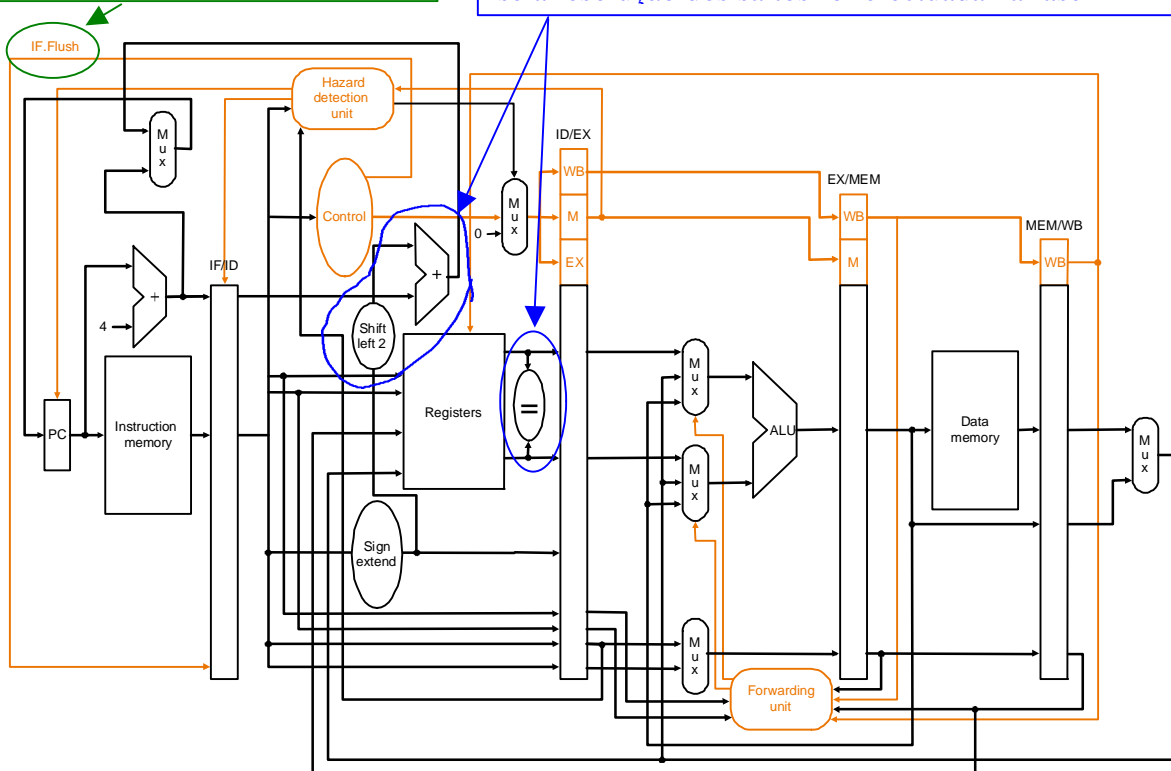
# Pipelining

- Resolução de dependências controlo
  - Os saltos condicionais implicam uma penalização de 3 ciclos, uma vez que o endereço de salto apenas é resolvido na fase MEM:



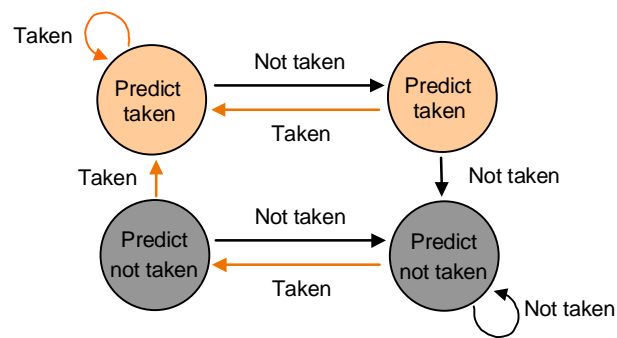
*IF.Flush* permite limpar o registo IF/ID, inserindo um *nop*

A penalização dos saltos pode ser reduzida a um ciclo se a resolução dos saltos for efectuada na fase ID

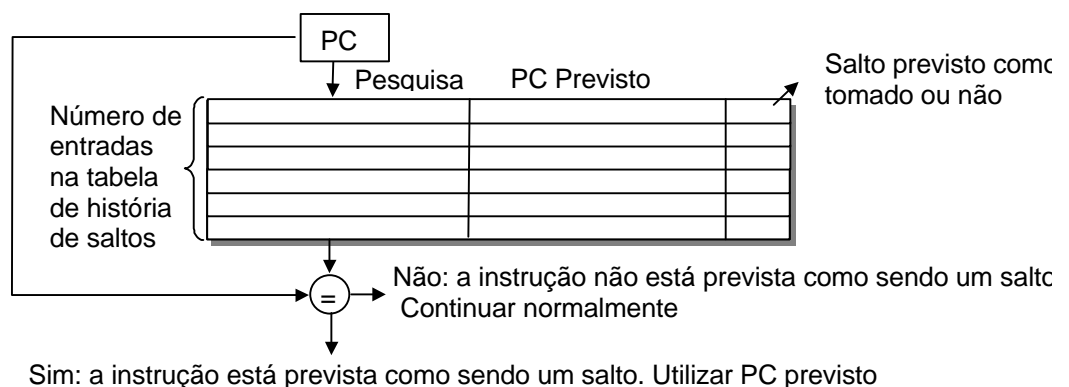


# Pipelining

- Resolução de dependências controlo
  - A previsão estática de saltos**, por exemplo, prevendo que o salto não é tomado pode ser implementada apenas adicionado lógica para eliminar as instruções em IF, ID, EX ( $\approx 50\%$  de previsões acertadas).
  - A previsão dinâmica de saltos** baseada numa tabela com a história dos saltos anteriores ( $\approx 90\%$  de previsões acertadas).
    - Um bit por endereço de salto é pouco eficiente, uma vez que nos ciclos *for* falha sempre duas vezes: na entrada (resultante da última execução) e na saída.
    - Esquema com dois bit apenas altera a previsão quando erra duas vezes, o que é mais eficiente:

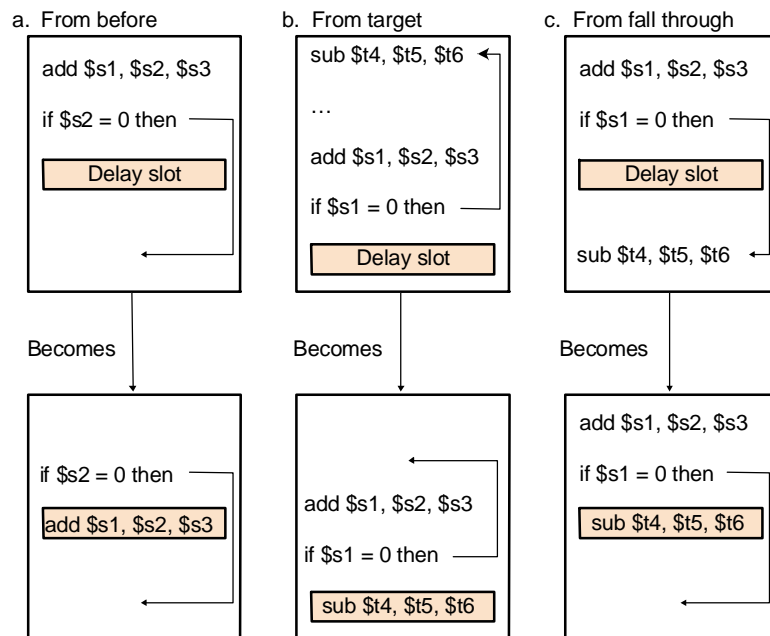


- A tabela com a história dos saltos** contém o endereço da instrução de salto e os bits correspondentes à informação sobre os últimos saltos realizados
- A previsão de saltos também requer uma **tabela com os endereços de salto tomados** anteriormente, uma vez que o endereço de salto só é calculado em EXE.
- As duas tabelas têm uma implementação idêntica às *caches* e podem-se juntar numa só:



# Pipelining

- Resolução de dependências controlo
  - Os saltos retardados**, executam sempre a instrução seguinte ao salto.



- Remove a penalização do salto sempre é encontrada uma instrução para preencher o slot ( $\approx 50\%$  dos casos).
- Os saltos retardados são uma solução limitada porque a sua eficiência diminui com a profundidade da *pipeline* e com a super-escalaridade.
- Desempenho da *pipeline* com *stalls*

$CPI_{pipeline} = CPI_{Ideal} + \text{ciclos de } stalls \text{ por instrução (CPI } stalls)$

$$Ganho = \frac{Texec_{sem\ pipeline}}{Texec_{com\ pipeline}} = \frac{CPI_{ideal} \times \#estágios \times Tcc_{sem\ pipeline}}{(CPI_{ideal} + CPI_{stalls}) \times Tcc_{pipeline}} = \frac{\#estágios}{1 + CPI_{stalls}}$$

- Exemplo: impacto da utilização de uma só porta para a memória

Máquina A – Memória *dual port*

Máquina B – Memória *single port*, relógio 1,05 vezes superior

40 % *Loads*

$$Ganho_B = \frac{Texec_A}{Texec_B} = \frac{(1+0) \times Tcc_A}{(1+0,4 \times 1) \times Tcc_B} = \frac{1 \times Tcc_A \times 1,05}{1,4 \times Tcc_A} = 0,75$$

**A Máquina B tem pior desempenho!**

# Pipelining

- Desempenho das variantes do processador MIPS

- Tempos acesso às unidades funcionais:

Memória (Leitura ou escrita)	ALU	Banco de registos
2 nanosegundos	2 nanosegundos	1 nanosegundo

- Qual a diferença de desempenho entre uma implementação com *pipelining*, uma implementação multi-ciclo e uma implementação *single-cycle* com ciclo de duração fixa para a seguinte mistura de instruções:

Tipo de Instrução	Frequência
Tipo R	49%
<i>Load</i>	22%
<i>Store</i>	11%
<i>Branch</i>	16%
<i>Jump</i>	2%

- Na versão com *pipeline* assume-se que 50% dos *loads* são utilizados na instrução seguinte, e que 75% dos saltos são previstos de forma correcta.

Tipo de Instrução	Busca da instrução	Leitura de registos	Operação na ALU	Acesso à Memória	Escrita em registo	Total
Tipo R	2	1	2		1	6 ns
<i>Load</i>	2	1	2	2	1	8 ns
<i>Store</i>	2	1	2	2		7 ns
<i>Branch</i>	2	1	2			5 ns
<i>Jump</i>	2					

- Single-cycle**

$$CPI=1$$

$$\text{duração de cada ciclo (Tcc)} = 8 \text{ ns}$$

- Multi-ciclo**

$$CPI = 0,49 \times 3 + 0,22 \times 5 + 0,11 \times 4 + 0,16 \times 3 + 0,02 \times 2 = 2,21$$

$$Tcc = 2 \text{ ns}$$

- Pipeline**

$$CPI_{load} = 0,5 \times 1 + 0,5 \times 2 = 1,5 \text{ (50\% originam stall)}$$

$$CPI_{branch} = 0,75 \times 1 + 0,25 \times 2 = 1,25 \text{ (25\% de previsões erradas)}$$

$$CPI = 0,49 \times 1 + 0,22 \times 1,5 + 0,11 \times 1 + 0,16 \times 1,25 + 0,02 \times 2 = 1,17$$

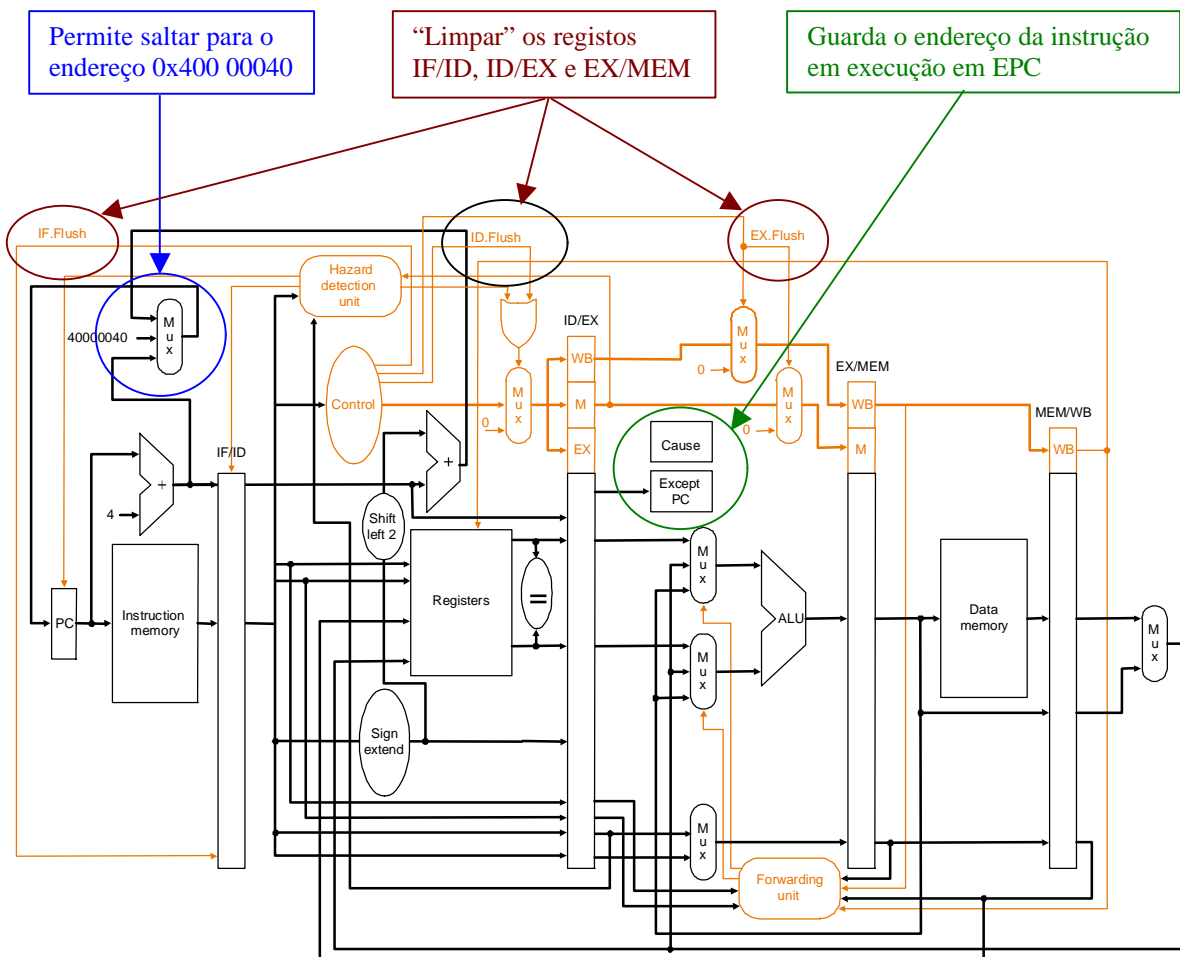
$$Tcc = 2 \text{ ns}$$

$$\text{Ganho Single-cycle/pipeline} = \#I \times 1 \times 8 / \#I \times 1,17 \times 2 = 3,42x$$

$$\text{Ganho Multi-ciclo/pipeline} = 2,21 / 1,17 = 1,89x$$

# Pipelining

- Suporte a excepções
  - Quando ocorre uma excepção o controlo é transferido para o programa no endereço de memória 0x4000040
  - É necessário “limpar” os registos da *pipeline* que contêm instruções parcialmente executadas, posteriores à instrução onde ocorre a excepção.
  - O valor actual do PC deve ser guardado em EPC
  - DP para suporte a uma excepção aritmética (por exemplo em *add*):



- O tratamento de excepções é complexo, uma vez que estas podem ocorrer em vários estágios simultaneamente, obrigando a um esquema de prioridades
- Algumas arquitecturas implementam excepções imprecisas, onde o EPC pode não indicar exactamente a instrução que provocou a excepção, deixando essa responsabilidade ao sistema operativo

## Bibliografia

- **Secções 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7** de *Computer Organization and Design: the hardware/software interface*, D. Patterson and J. Hennessy, Morgan Kaufmann, 2<sup>a</sup> edição, 1998.
- **Adicional:**
  - Secções 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 de *Computer Architecture: A Quantitative Approach*, J. Hennessy and D. Patterson, Morgan Kaufmann, 2<sup>a</sup> edição, 1996.